VRIJE UNIVERSITEIT BRUSSEL

ENVIRONMENTAL PROGRAMMING ASSIGNMENT

Antarctic melt

BESUAL GROUP

Alfred Otom

Sudam Selaka Samarasinghe

Beryl A. Omollo

January 25, 2025

Table of Contents

List of Figures

1. Introduction

Accurate prediction of global sea-level rise has become increasingly critical in the face of climate change and escalating environmental pollution. This can be achieved by understanding the current and recent glacier melting along the Antarctic ice sheet coastal margins. It is postulated that climate change especially higher temperatures cause melting of the ice-sheets thus, influencing the amount of melt. This project analyzed daily surface melting data from 1979- 2023 to evaluate ice-sheet melting patterns in the recent years. In this project, we developed software designed to evaluate ice-sheet melting in Antarctica. Users can specify a time period of interest to calculate the number of melting days and the extent of ice melt, enabling them to analyze melting trends and gain insights into the contribution of climate change to this phenomenon. Additionally, we developed a graphical user interface (GUI) to interactively execute the code.

2. Datasets

We used two datasets NetCDF (Network Common Data Form) and iceshelves shapefile data. NetCDF file is a multi-dimensional labeled data, and in this project, it contained the daily surface melting data for every pixel from 1979-2023 at a resolution of 25x25km. The pixels were labelled as 0, 1 and -10 representing no melt, melt, and no available data respectively. Additionally, the shapefile had outlines of Antarctic Ice shelves. These datasets are available at Antarctic Polar Stereographic projection (EPSG:3031).

3. Coding

3.1 Assignment Tasks

Task 1: Reading the iceshelves shapefile (.shp)

Geopandas module was imported using the code **import geopandas as gpd**. The geopandas library provides necessary tools for reading geospatial pandas data frame. Thereafter we created file path before reading the data using the

code below:

shapefile = 'env_pro_data/IceShelf_Antarctica_v02.shp'

gdf = gpd.read_file(shapefile)

Task 2: Reading the melt dataset (NetCDF; .nc)

The project used Xarray module to read the NetCDF file. Xarray is a robust library for handling multi-dimensional data, such as netCDF files. It offers an easy-to-use method for accessing, modifying, and analyzing data. Import the xarray module using *import xarray as xr* then open the netCDF file;
data = xr.open_dataset('env_pro_data/CumJour-Antarctic-ssmi-1979-2023-H19.nc'). This code opens the netCDF file and loads it into an xarray dataset object.

Task 3: Calculation of the cumulative number of melt days per year for each pixel, for each year

First, we defined the time period using the following assignment operator;
start_year = int(input("Enter the start year: "))
end_year = int(input("Enter the end year: "))
This helps to set the temporal boundaries in the analysis and visualizations.
Next, we filtered the dataset along the time dimension using .sel() function.
filtered_data = data.sel(time=slice(f"{start_year}-01-01", f"{(end_year)}-12-31"))
The slice () specifies the time values. It is important to note that subsequent analysis will use the filtered data above.

Before calculating the cumulative number of melt days, we grouped the data by year using .groupby() to ensure that each year is treated separately for the cumulative calculation. The .sum() calculates the cumulative number of melt days per year.

melt_year = melt.groupby('time.year').sum(dim='time')

Task 4: Calculation of the average cumulative annual melt days over the total time period

In this task we first calculated average cumulative melt then plotted the results. The average cumulative annual melt using .mean(dim **='year', skipna=True**). Skipna condition ignores the non-values during mean calculation. The .where() function is a conditional factor which is used to replace zeros with nans.

```
avg_melt_days = melt_year.mean(dim='year', skipna=True)
avg_melt_days = avg_melt_days.where(avg_melt_days!= 0, np.nan)
```

For plotting, we imported necessary libraries including; cartopy and matplotlib. Afterwards, we defined and called the plotting function using the code below:

```
plot(gdf, lon, lat, avg_melt_days,title=f'Annual Average of
Cumulative Melt Days ({start_year}-
{end_year})',vmax=math.ceil(avg_melt_days.max() / 10) *
10,vmin=math.floor(avg_melt_days.min() / 10) *
10,save_name=f'Annual_Average_of_Cumulative_Melt_Days_{start_
year}_{end_year}.png')
```

The above code plots the map of Antarctica showing the annual average melt days over the specified time period as shown in Figure 1 below

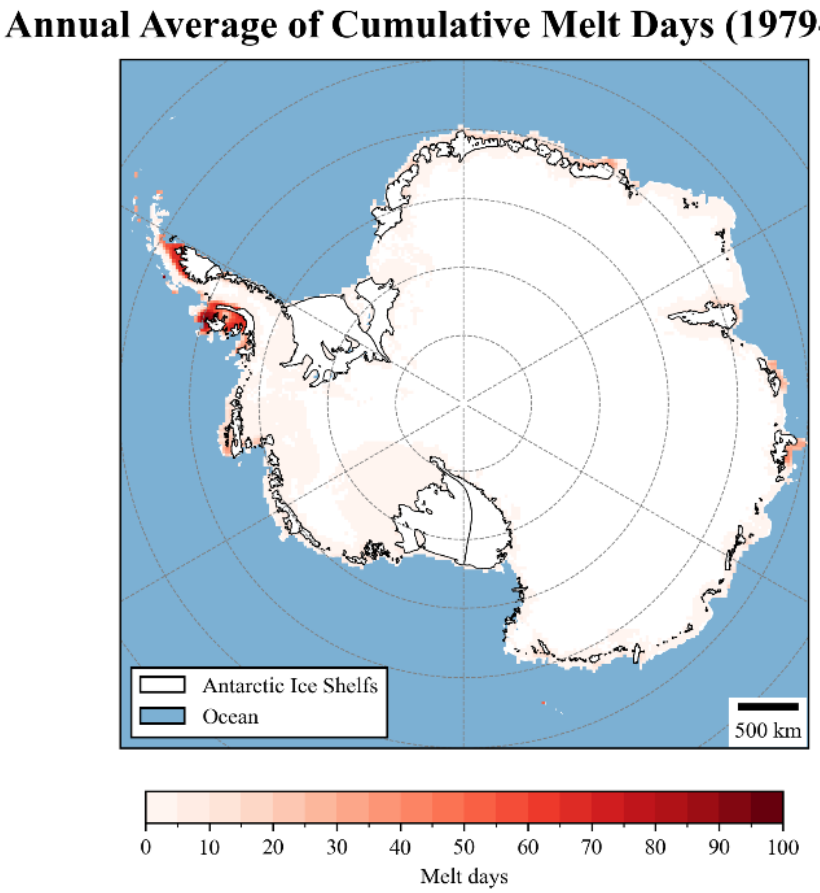

Figure 1: Annual average melt days from 1979-2023

Task 5: Calculation of maximum melt days and maximum melt extent

This section of the code involved two major tasks that include calculation of: most melt days and largest melt extent

The xarray, numpy package and the functions in pandas were called, while for plotting, geopandas, matplotlib and cartopy. Calculation of the melt days in all the pixels on every year from the start date to the end date of the data, then picking the year that had the highest number of days exhibiting melting. The second task involved determination of the melting of each pixel and then aggregating to identify the section with the widest area exhibiting melting over the entire data period.

- **max_melt_days=melt_year.max(dim=['y', 'x'] skipna=True).to_dataframe(name='Max_melt_days')**

Calling the function .max(dim=['y', 'x'] calculates the maximum melt days in

the entire area going through each pixel in the dimensions of longitude (x) and latitude (y) across the entire area for the period ranging from 1979 to 2023. While doing so, the skip=True is called to ensure only melt data is considered while ignoring nonvalues and zero melt. The data is the stored in a pandas data frame with a column name Max_melt_days. The function .max() scans the through the created dataframe column and identifies the greatest value for storage.

max_melt_days['Max_melt_days'] == max_melt_days_value:

Once the maximum melt day value has been identified, the above code compares each value in the column of the created data frame with the maximum value to ascertain whether it is higher which yields (True) or lower which yields (False) creating a Boolean, thereafter only retaining True conditions.

- **max_melt_days_years = max_melt_days[max_melt_days['Max_melt_days'] == max_melt_days_value].index.tolist()**

The .index then helps to retrieve the years meeting the true condition and tolist() is used for conversion of time index (year) into list.

- **print(f'Max Melt days is {int(max_melt_days_value)} in the following year/years: {max_melt_days_years}')**

The **f-string** is then called to store maximum melt value and the year in which the melt occurred and the **int(max_melt_days_value)** converts the float value into an integer.

- **melt_year_filtered = melt_year.where(melt_year != 0)**
- **melt_extent = melt_year_filtered.notnull().sum(dim=['y', 'x'])**

For largest melt extent, the function **melt_year.where(melt_year != 0),** ensures that through the entire dataframe, only those years that are nan zeros are considered and stored in **melt_year_filtered,** thus considering only the relevant data, filtering out non melt pixels. The function **.notnull()** iterates through the data thereby returning a boolean array of True for actual melt (non-NaN data) and False for NaN values (invalid data).Calling **.sum(dim=['y', 'x'])** leads to addition of all the **True** values across the spatial

longitudinal and latitudinal extent for each of the years under consideration.

- **max_melt_extent_value = melt_extent.max()**
- **max_melt_extent_years=melt_extent.sel(year=melt_extent== max_melt_extent_value)**

The line of code, **max_melt_extent_value = melt_extent.max()** returns the year where there was the greatest melt. The code line **max_melt_extent_years = melt_extent.sel(year=melt_extent == max_melt_extent_value),** creates a Boolean that checks what years meets the criteria of maximum melt and then select them out for printing. The **f-string** is then called for the storage of the year with largest melt extent while converting the float result into an integer using the **int(max_melt_extent_value)**.
{",".join(map(str, max_melt_extent_years.coords["year"].values))}'. Extracts the years where maximum melt occurred as integers and converts into a string before joining using comma punctuation mark and space.

Task 6: Plotting the images generated in task 5

```
for year in max_melt_days_years:
    tmp = melt_year.sel(year=year)
    tmp = tmp.where(tmp != 0, np.nan)
```

The code line iterates through the list created for the maximum melt days in given years. It then creates a temporary file called **tmp = melt_year.sel(year=year)**: by slicing the data and only picking the year with maximum melt. The line **tmp = tmp.where(tmp != 0, np.nan)** is used to only select value that are not equal to zero, denoting melting while at the same time equating non melt to nan and modifies the temporary file.

- **plot(gdf, lon, lat, tmp,title=f'Max Melt days is {int(max_melt_days_value)} in {year}',vmax=math.ceil(tmp.max() / 10) * 10,vmin=math.floor(tmp.min() / 10)**

The plot function then calls gepandas dataframe (gdf) together with the corresponding longitude and latitude for each grid and iterates through the entire spatial extent combining with melt data created in temporary file (tmp). The plot title is generated through a dynamically (**title=f'Max Melt**

days is {int(max_melt_days_value)}). The minimum (**vmin=math.floor(tmp.min() / 10)**) and maximum (**vmax=math.ceil(tmp.max() / 10) * 10**) melt data for the year (**{year}**) is rounded off to the nearest ten. This then prints two plots for the corresponding years with maximum melts (230) namely in 1989 and 2017 as 230 as Figure two a 1989 and b 2017 respectively.

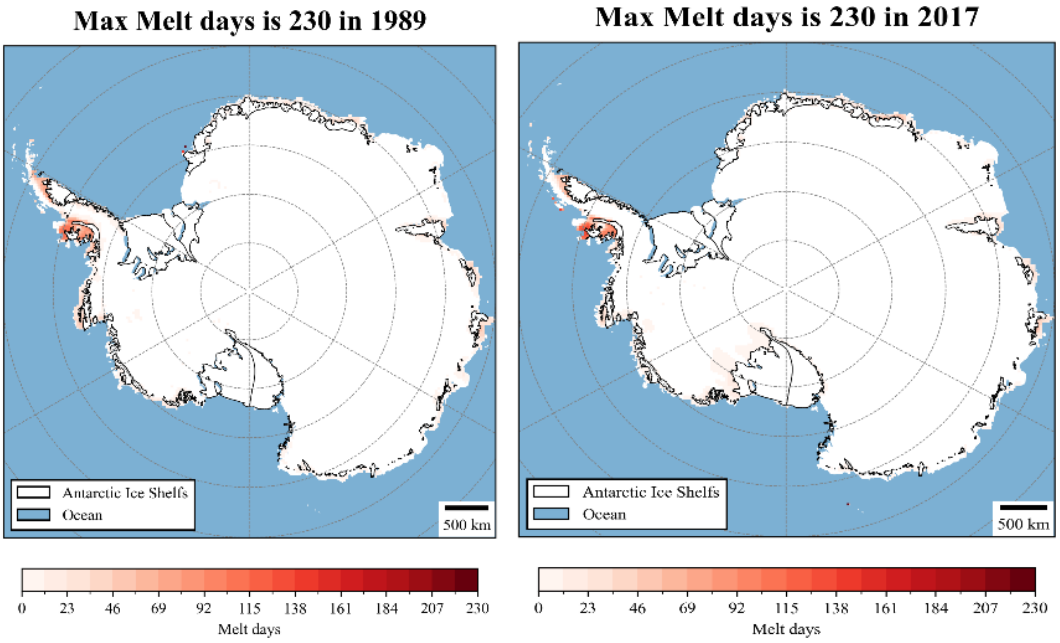

Figure 2: Maximum melt in 1989 (left) and maximum melt in 2017 (right)

The creation of the difference image of average melt days and the year with maximum melt days employs the reuse of the earlier code used for plotting the maximum melt days years as shown below;

```
for year in max_melt_days_years:  # Iterate through the years in max_melt_days_years
    tmp = melt_year.sel(year=year)
    tmp = tmp.where(tmp != 0, np.nan)
```

Except the part of the code that is used for generation of the temporary file that uses the difference to deduct the average melt days and maximum melt days to give the difference as outlined by the code **difference = avg_melt_days - tmp.** It is then the difference that is plotted as Figure 4 both a and be denoting corresponding difference between average melt days and

the particular year with maximum melt as calculated earlier.

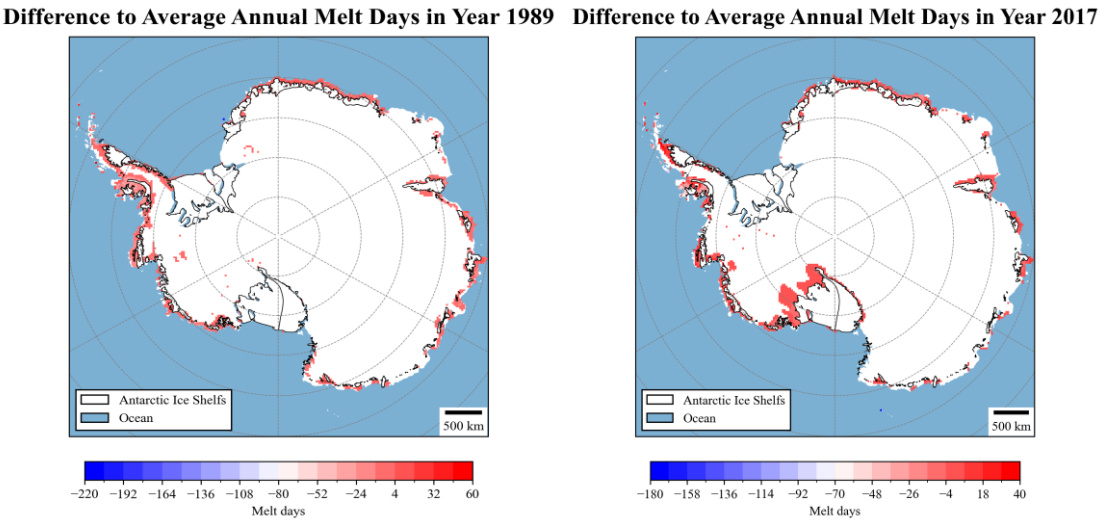

Figure 3: Difference in average annual melt and the year with maximum melt corresponding to 1989 (left) and 2017 (right) respectively

The plot for the largest extent of melt begins by extraction of the year with the maximum melt year from the data column of the dataframe **max_melt_extent_years** using **.values** attribute extracts the underlying value while **.item()** ensures only one year is picked. The earlier code for selection of the ideal data while omitting the non-melt areas is then used.

- **tmp = melt_year.sel(year=year_value)**
- **tmp = tmp.where(tmp != 0, np.nan)**

Finally the data is plotted calling the geopandas dataframe (gdf), latitude and longitude of the relevant grids and mapping with the created temporary fikle containing the melt extent as shown in Figure 4.

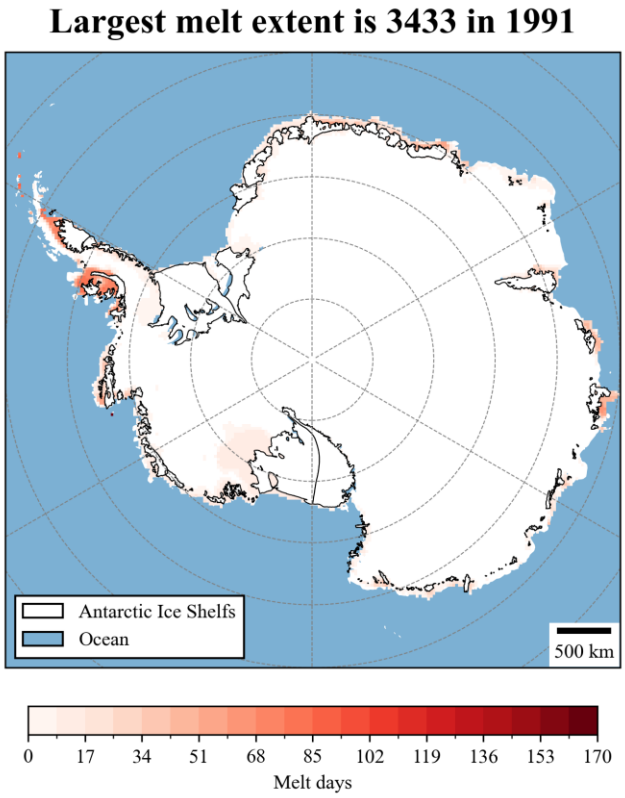

Figure 4: Largest melt extent experienced in 1991

Task 7: Convert your xarray to a Pandas DataFrame

The pandas and numpy package were called to manipulate the data into the desired dataframes and attributes.

- **melt_data_flattened = melt.values.reshape(melt.shape[0], -1)**

The function **(melt.shape[0], -1)** ensures all the dimensions of the melt dataframe is collapsed (longitude and latitude) except for time is preserved. The library function **.reshape()** returns the data without alterations, while **melt.values** loops into the Numpy array to extract the data that is then stored in the **melt_data_flattened** as two dimensional format (2D) containing only time and combining longitude and latitude.

- **melt_extent = np.nansum(melt_data_flattened > 0, axis=1)**

The **axis=1** implies the function to be executed is on the second axis (column) looping on rows of the data array and **melt_data_flattened > 0** iterates through each element and returns a Boolean of **True** where the element is greater than 0 and **False** where it is less than zero. Then **np.nansum**

computes the summation of all non NaN values to stored in **melt_extent.**

- **days_of_year = time.dt.dayofyear**

Extraction of datetime from the dataframe is carried our by calling **.dt** while recognizing all the days of the year including in a leap year through **.dayofyear** thereby running up to 366th day**.** The output is an array of integers stored in the **days_of_year.**

- **df_melt_extent=pd.DataFrame({"datetime":time['time'],"day_of_year":days_of_year,"melt_extent": melt_extent})**

The code creates a pandas DataFrame named **df_melt_extent** that has got three dimensional columns: **datetime** obtained from **[‘time’]** which exists in the dataframe, **day_of_year** that was calculated and stored in **days_of_year** and the **melt_extent** stored in each iteration of time stamp, the result then printed as pandas dataframe using **print(df_melt_extent).**

Task 8: Plot melt extent timeseries

The applied libraries include pandas, numpy

- **df_melt_extent['year_group'] = df_melt_extent['datetime'].apply(lambda x: x.year if x.month >= 6 else x.year - 1)**

To plot the years running from June of the current year to May of the subsequent year, there is need to create a column with the name **['year_group'].** This new column is attached to the **df_melt_extent.** The function **.apply()** iterates through the elements of the rows while calling **lambda** on each time stamp to add the **melt_extent** at each **[‘datetime’].** This it does by considering the value of melt of the current year **x.year** if the month date is equal to or greater than six that is June and beyond **x.month >= 6** otherwise it considers the previous year **else x.year -1.**

- **df_melt_extent = df_melt_extent.loc[(df_melt_extent['year_group'] >= start_year) & (df_melt_extent['year_group'] <= end_year)]**

The function is meant to return a Boolean when the **True** and **False** condition arise to meet the scenario for checking which part of the year in the **df_melt_extent['year_group'].** Depending on datetime applicable, then the

operators **>= start_year** and **<= end_year** are called to check whether the condition merit application of which part of the year, current or previous respectively. An operator **&** is applied to ensure both conditions are true for execution to hold.

def adjust_date(date):

- **if date.month >= 6:**

return pd.Timestamp(year=1979, month=date.month, day=date.day)

else:

return pd.Timestamp(year=1980, month=date.month, day=date.day)

Calling the above function expects an input from the keyboard for the start date upon which it checks whether the date is June or beyond upon which it returns a time step of the year **(year=1979, month=date.month, day=date.day)**, the month and the day if not, the subsequent year is picked with month and date **(year=1980, month=date.month, day=date.day)**.

- **df_melt_extent = df_melt_extent.copy()**

The code is meant to create a **copy** of the **df_melt_extent** dataframe to ensure there is no alterations to the original **df_melt_extent**.

- **df_melt_extent['custom_date'] = df_melt_extent['datetime'].apply(adjust_date)**

The code **['custom_date']** creates a column named custom_date on the dataframe **df_melt_extent,** obtaining the elements through **['datetime'].** The part **.apply(adjust_date)** ensures the months are aligned to the current year or subsequent depending on the result of the iteration.

- **average_extent = df_melt_extent.groupby('custom_date')['melt_extent'].mean()**

A new dataframe is created by the above code by first ensuring the **df_melt_extent** accessed is accessed and aggregated based on the **.groupby('custom_date') then the** function **.mean()** facilitates averaging based on the groups. The result is then plotted on the same figure for all the years as shown in Figure 5 with black solid line**.**

- **max_melt_extent_year_value = int(max_melt_extent_years['year'].values[0])**

A Numpy array is created by extracting the first value using **.values[0])** from the column named years by calling the function **['year']** and finally converting the result into an integer using **int().**

- **for year, group in df_melt_extent.groupby('year_group'):**

color = 'red' if year == max_melt_extent_year_value else 'grey'

alpha = 1.0 if year == max_melt_extent_year_value else 0.5

label = f"Max Melt Extent ({year})" if color == 'red' else None

plt.plot(group['custom_date'], group['melt_extent'], color=color,

alpha=alpha, label=label)

Considering the column '**year_group**' in the dataframe **df_melt_extent,** consideration is made by iteration to pick the year with the maximum melt **(if year == max_melt_extent_year_value**) and it is highlighted **red** if it is the greatest otherwise plotted **grey**. The code then applies **f-string** to save the output and label as Max Melt Extent attaching the exact year **{year}** or else left unnamed**.**

- **plt.plot(average_extent.index, average_extent.values, color='black', linewidth=2, label=f'Average Melt Extent for {start_year}-{end_year} period')**

The plotting involves calling the average extent dataframe picking independent values as x-axis denoted by **average_extent.index** while the dependent variable being the y-axis as **average_extent.values** and it is indicated in the plot by color black line. The **f'Average Melt Extent for {start_year}-{end_year}** is applied using **f-string** to name the line spanning through the entire dataset.

- **plt.title(f'Melt Extent Data for {start_year}-{end_year} period')**
- **plt.ylabel('Melt Extent / (Pixels)')**
- **plt.grid(True, linestyle='--', alpha=0.6)**

The code for adding the title employs the **f-string** to save beginning

{start_year} and end date {end_year} period separated by dash **(-).** The y-axis label is also added **('Melt Extent / (Pixels)'**

- **plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%b'))**
- **plt.gca().xaxis.set_major_locator(mdates.MonthLocator())**
- **plt.legend()**

The block code ensures that the x-axis is plotted by accessing the current axes **plt.gca(). At** the same time displaying the tick labels **.xaxis.set_major_formatter()** while showing the months of the year **formatter(mdates.DateFormatter('%b'))** and aligning the tick labels to the months of the year **locator(mdates.MonthLocator()).**

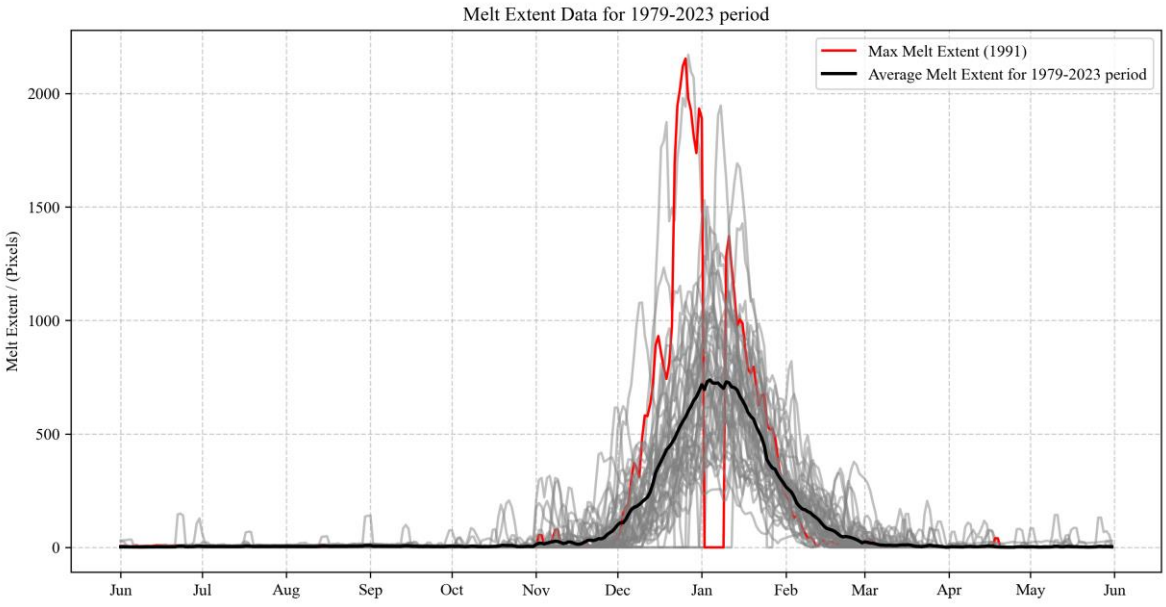

Figure 5: Melt extent over the years (grey), maximum melt extent experienced in 1991 (red) and the average melt extent across the entire data frame (black)

Task 9: Fifteen (15) largest ice shelves and average melt extent

The following section of the code employed pandas, geopandas, rioxarray and matplotlib libraries.

- **gdf = gdf.to_crs(epsg=3031)**
- **sorted_gdf = gdf.sort_values("area_calculated", ascending=False**
- **top_15_gdf = sorted_gdf.head(15)**

A geopandas dataframe **(gdf)** is created by calling **.to_crs()** which creates a coordinate reference system representing the Antarctica Polar Stereographic **(epsg=3031)** projection to ensure there is no shape deformation of the

georeferenced data. Upon creating the dataframe, area of each pixel is calculated using **gdf.geometry.area** and stored in **gdf["area_calculated"]**, starting with the largest in a descending order **(ascending=False)**. After creating the dataframe in a descending order, the first largest fifteen (15) are picked out **sorted_gdf.head(15)** and stored in **top_15_gdf.**

- **melt = melt.rio.write_crs("EPSG:3031", inplace=True)**

The invoking of the function **inplace=True**, ensures that the operation occurs on the **melt** data directly by linking Antarctica Polar Projection **EPSG=3031** on melt data via to **.rio.write_crs()** extension. Outputs are then stored for subsequent manipulations through **results = [].**

- **for area_name in top_15_gdf['NAME']:**

The iteration through the area names of the top 15 largest melt are then assigned leading to a pandas series bearing the names of the top 15 melt areas (**top_15_gdf['NAME']:**).

- **region = top_15_gdf[top_15_gdf["NAME"] == area_name]**

Only the top fifteen largest areas are extracted **top_15_gdf["NAME"]** to create a subset of rows 15 in number and the part of the code checks if the area tallies with corresponding name and only picks Boolean outcome **True** through the operation **[top_15_gdf ["NAME"] == area_name]** and stored in **region.**

- **clipped = melt.rio.clip(region.geometry, region.crs)**

The **melt** dataset is trimmed using **.rio.clip(region.geometry, region.crs),** which ensures that the **region.geometry** column is the only one under consideration and is transformed to appropriate coordinate reference system through **region.crs**.

- **melt_masked = clipped.where(clipped == 1)**

A dataset **melt_masked** is created by iteration through the raster dataset called **clipped** by operation **.where(clipped == 1)** and only returning cases in which the clip returns **True** which is 1 and where the condition is not met **False** masked to NaN.

- **yearly_average_melt = melt_masked.resample(time="1YE").mean()**

- **yearly_sum = yearly_average_melt.sum(dim=["x", "y"])**

An annual average melt (**yearly_average_melt**) is created through, checking the time dimension of the data, **.resample(time="1YE")** returns annual frequency **"1YE"** and then an average is calculated on the frequencies **.mean()**. A summation which is one dimensional (**yearly_sum**) of all the annual means (yearly_average_melt) is then obtained through the entire coordinate system **dim= ["x", "y"]** reduced from raster data to two dimensional array then obtains the sum through **.sum()**

- **df_yearly_sum = yearly_sum.to_pandas()**
- **average_yearly_sum = df_yearly_sum.mean()**
- **results.append({'NAME': area_name, 'average_yearly_melt_extent': average_yearly_sum})**
- **df_top_15_melt = pd.DataFrame(results)**
- **print(df_top_15_melt)**
- **df_top_15_melt.to_csv(f'df_top_15_melt_{start_year}to{end_year}.csv', index=False)**

The one dimensional (yearly_sum) is used to create an additional dataset of pandas dataframe format (**df_yearly_sum**) through calling the function **.to_pandas().** Further, a mean is obtained of the **df_yearly_sum** through applying **.mean()** to create **average_yearly_sum**. The **.append()** adds the the strings area name **('area_name')** and it's corresponding mean annual melt extent '**average_yearly_melt_extent**' and creates a new pandas dataframe named **df_top_15_melt.**

On the newly created pandas dataframe, we apply **.to_csv()** to export it as an excel in csv format (**.csv**) retaining the attributes of year of commencement **{start_year}** through to the year of ending **{end_year}** while omitting the row index **index=False.**

- **plt.pie(df_top_15_melt['average_yearly_melt_extent'], labels=df_top_15_melt['NAME'], autopct='%1.1f%%', startangle=90)**
- **plt.title(f'Average Melting Extent for the Largest 15 Ice Shelves from {start_year} to {end_year}', pad=20)**
- **plt.axis('equal')**
- **plt.savefig(f'Average Melting Extent for the Largest 15 Ice Shelves from {start_year} to {end_year}.png', bbox_inches='tight')**

The plot in form of a pie chat is obtained calling the **'average_yearly_melt_extent'],** attaching corresponding name of the region as a label **['NAME']** and restricting the start angle of the pie chat at 90 degrees **(startangle=90)** as opposed to the default zero degrees value. Thereafter chat title (**f'Average Melting Extent for the Largest 15 Ice Shelves from {start_year} to {end_year}'**) is given using **f-string** and plotted while maintaining the aspects of the image ratio **.axis('equal')** and saved as **.savefig().**

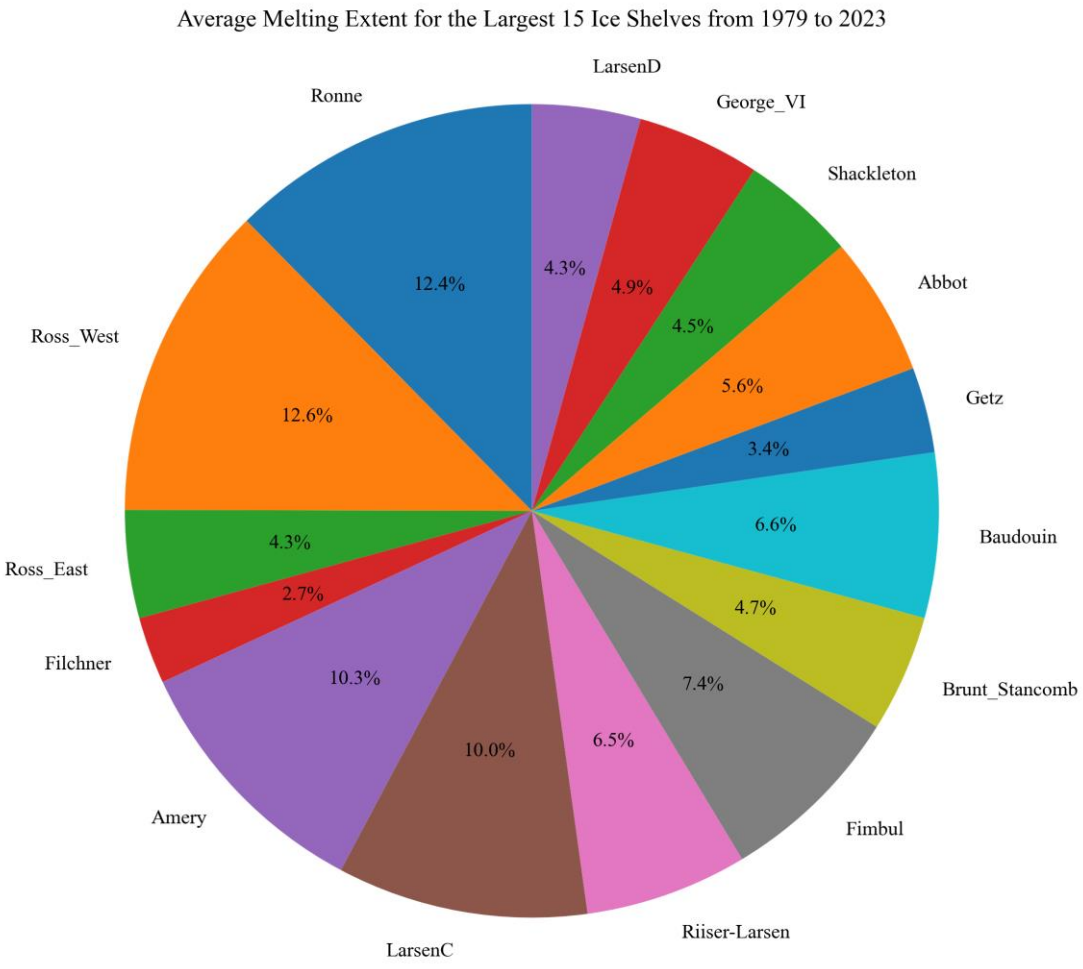

Figure 6: Average melting extent for the largest 15 ice shelves from 1979-2023

3.2 Flow chart

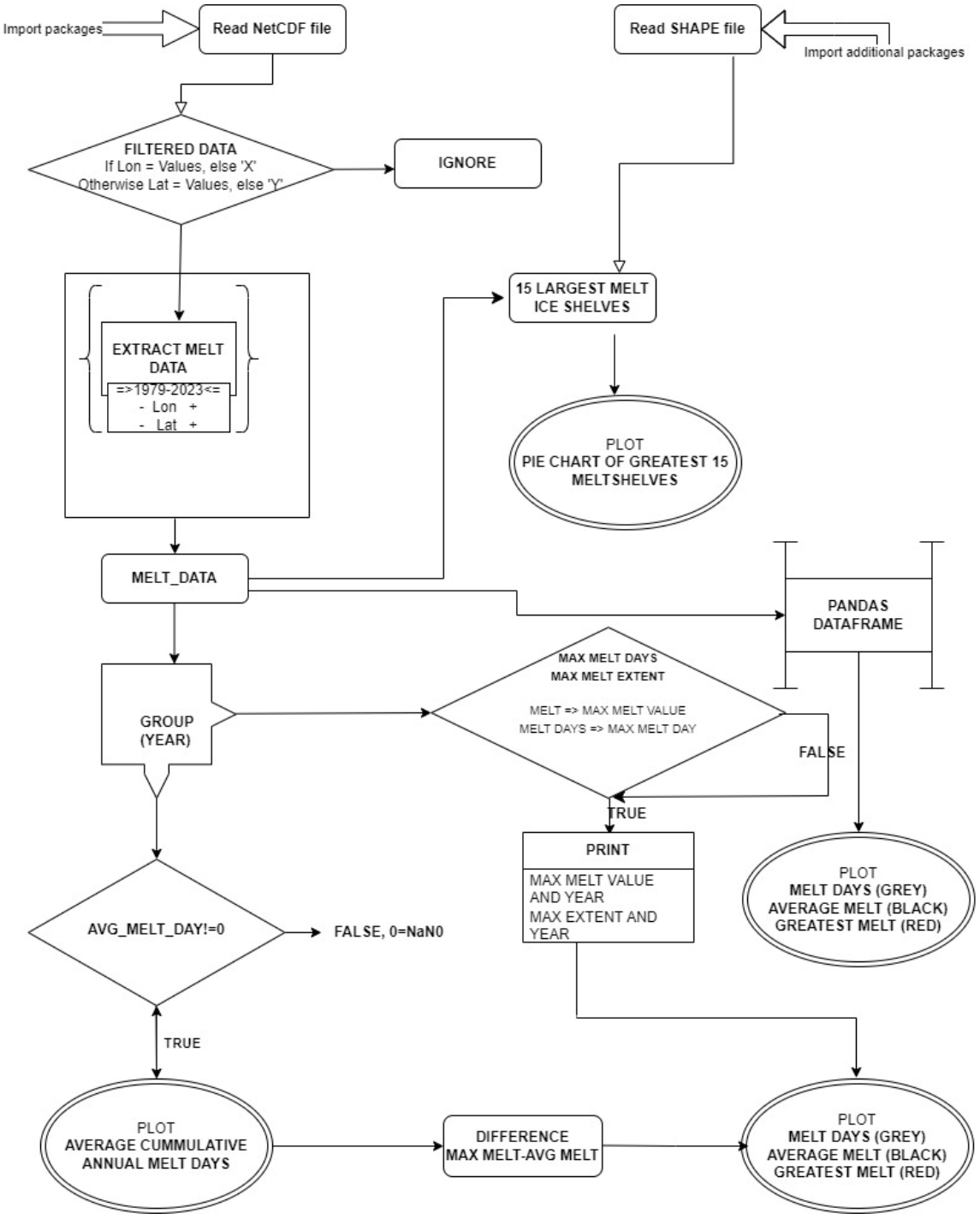

Figure 7: A flowchart showing code development process

4. Graphical User Interface (GUI)

The BESUAL Melt Data Analyzer is a user-friendly Graphical User Interface (GUI) application designed for analyzing and visualizing cumulative melt data from Antarctic regions. The software enables researchers and environmental scientists to work seamlessly with NetCDF datasets and shapefiles to derive meaningful insights about melt patterns and trends over specified time periods. This tool was developed to support decision-making and enhance understanding of Antarctic melt dynamics based on the code we developed in the previous section.

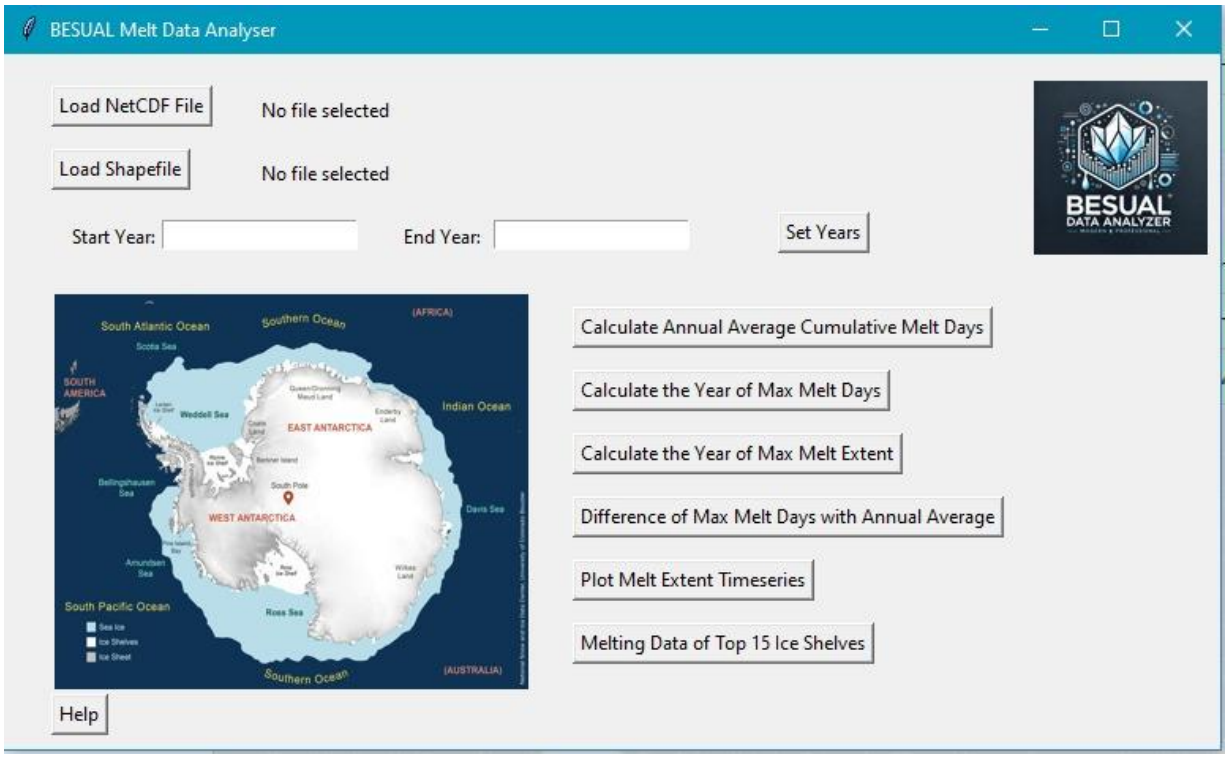

Figure 8: Graphical User Interface

Key Functionalities

1. Data Loading:

The application supports loading of two primary data types:

NetCDF Files: Contains cumulative melt data for Antarctic regions.

Shapefiles: Provides spatial boundaries and regions of interest for mapping and analysis.

2. Time Range Selection:

Users can define custom start and end years to focus analyses on specific time periods. The software ensures the selected years fall within the dataset's available range and notifies users of any discrepancies. This feature enables the user to carry out the scenario analysis with different time periods.

3. Comprehensive Analysis Tools:

The software includes several pre-defined analytical functions same as in the basic code we developed.

Annual Average Melt Days: Calculates and visualizes the mean annual cumulative melt days over the selected time range.

Year of Maximum Melt Days: Identifies and plots the year with the highest cumulative melt days.

Year of Maximum Melt Extent: Determines the year with the largest melt extent (area affected by melting) and visualizes the data.

Difference Analysis: Compares the annual average melt days with the melt days of the year with the maximum melt days to highlight anomalies.

Melt Extent Timeseries: Displays trends in melt extent over the selected years, including individual year data and an aggregated average. The curve of maximum melting days year is highlighted in Red.

Top 15 Ice Shelves Analysis: Automatically analyzes the melt patterns for the 15 largest ice shelves, providing insights into their vulnerability to ice shelve melt. This gives an idea about how the climate change affects the biggest ice shelves in Antarctica.

4. Intuitive Visualization:

The software offers high-quality visualizations using the Cartopy and Matplotlib libraries. Features include:

- Color-coded heatmaps to represent melt data.
- Interactive legends, scale bars, and gridlines for easy interpretation.
- Customizable color schemes and titles for enhanced presentation.
- Plots are saved automatically with proper name for easy late reference.

5. Data Export:

All visualizations can be saved as high-resolution images, making it easier to include them in reports, publications, or presentations.

The data frames developed in the ice shelf analysis also saved as a comma separated value file for later reference.

5. Software Installation Guide

The BESUAL Melt Data Analyzer (Basic script and GUI) requires a Python virtual environment to ensure a clean and isolated setup for running the software. Key dependencies include packages like xarray, geopandas, matplotlib, cartopy, pandas, and numpy, alongside Python's built-in tkinter module for GUI development. Users can quickly set up the environment by creating a virtual environment, activating it, and installing dependencies listed in the provided requirements.txt file.

To run the scripts, please copy all the files in the zip folder named besual_submission.zip and run besual_script.py file for the basic script and besual_gui.py for the GUI program.

6. Developed Scripts

6.1 Basic Script

```
# Introducing the packages

# Array package
import numpy as np
import pandas as pd
import xarray as xr
# For mathematical functions
import math
# Shapefile loading package
import geopandas as gpd
# Projection plot package
import cartopy as crs
import cartopy.crs as ccrs
from cartopy.feature import ShapelyFeature
# Ploting package
import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
from matplotlib.patches import Patch
from matplotlib_scalebar.scalebar import ScaleBar
import matplotlib.dates as mdates

# Defining the plotting function
def plot(shapefile, lon, lat, data, **kwargs):

    # Change the font type as Times New Roman
    plt.rcParams['font.family'] = 'Times New Roman'

    # Loading keywords
    title = kwargs.get('title',None)
    vmax  = kwargs.get('vmax',None)
    vmin  = kwargs.get('vmin',None)
    cmap  = kwargs.get('cmap','Reds')
    save_name = kwargs.get('save_name', None)

    # This is to adjust the interval of value.
```

```
    num_levels = 20
    levels = np.linspace(vmin, vmax, num_levels+1)
    norm = BoundaryNorm(levels, ncolors=256)

    # Ploting
    fig = plt.figure(figsize=(6,6),dpi=300)
    # According to the requirement of assignment, the projection (EPSG:3031)
is set.
    ax = fig.add_subplot(111, projection=ccrs.SouthPolarStereo())
    # Set mapping print out range
    ax.set_extent([-180, 180, -90, -65], ccrs.PlateCarree())

    # Make ocean color
    ax.add_feature(crs.feature.OCEAN, facecolor='#7cb0d3')
    ax.gridlines(ls='--',color='grey',lw=0.5)

    # Applying the shapefile to map
    shape_feature = ShapelyFeature(shapefile.geometry,
ccrs.SouthPolarStereo(), edgecolor='k', facecolor='white',lw=0.5)
    ax.add_feature(shape_feature)

    cs = ax.pcolormesh(lon, lat, data, norm=norm,
cmap=cmap,transform=crs.crs.PlateCarree())
    cb = fig.colorbar(cs,
ticks=np.linspace(vmin,vmax,11),orientation='horizontal', shrink=0.7,
pad=0.05)
    cb.ax.set_xlabel('Melt days',fontsize=10)
    ax.set_title(title,weight='bold',fontsize=18,pad=10)

    # Plot legend for coastline
    # plt.plot([-59.9,-60],[-59.9,-
60],c='k',label='Coastline',transform=crs.crs.PlateCarree())
    # plt.legend(fontsize=10)
    continent_patch = Patch(facecolor='white', edgecolor='black',
label='Antarctic Ice Shelfs')
    sea_patch = Patch(facecolor='#7cb0d3', edgecolor='black', label='Ocean')
    plt.legend(handles=[continent_patch, sea_patch], fontsize=10, loc='lower
left', frameon=True, framealpha=1, edgecolor='black', fancybox=False)
```

```
    # Add the scale bar
    scalebar = ScaleBar(1, location='lower right', length_fraction=0.1)  # 1
unit = 1 degree
    ax.add_artist(scalebar)

    if save_name:
        fig.savefig(save_name, dpi=300, bbox_inches='tight')

    plt.tight_layout()
    plt.show()
    plt.close()

# Task 1: Load the shapefile
shapefile = 'env_pro_data/IceShelf_Antarctica_v02.shp'
gdf = gpd.read_file(shapefile)

# Task 2: Load the NetCDF file
data = xr.open_dataset('env_pro_data/CumJour-Antarctic-ssmi-1979-2023-
H19.nc')

# Task 3: Extract data and preprocess with user-defined time period
# Define the preferred time period (e.g., from 1979 to 2023)
start_year = int(input("Enter the start year: "))
end_year = int(input("Enter the end year: "))

# Filter the dataset for the preferred time period
filtered_data = data.sel(time=slice(f"{start_year}-01-01", f"{(end_year)}-12-
31"))

# Extract filtered data
lon = filtered_data['lon'].values #if 'lon' in filtered_data else
filtered_data['x'].values
lat = filtered_data['lat'].values #if 'lat' in filtered_data else
filtered_data['y'].values
melt = filtered_data['melt']
time = filtered_data['time']

# Group data by year within the preferred time period
```

```
melt_year = melt.groupby('time.year').sum(dim='time')

# Task 4: Annual average and maximum cumulative melt days
avg_melt_days = melt_year.mean(dim='year', skipna=True) # to ignore Nan
values, if False; it would return NaN if any value along the dimension is NaN
avg_melt_days = avg_melt_days.where(avg_melt_days != 0, np.nan)  # Replace
zeros with NaN

# Plot the annual average
plot(gdf, lon, lat, avg_melt_days,title=f'Annual Average of Cumulative Melt
Days ({start_year}-{end_year})',vmax=math.ceil(avg_melt_days.max() / 10) *
10,vmin=math.floor(avg_melt_days.min() / 10) *
10,save_name=f'Annual_Average_of_Cumulative_Melt_Days_{start_year}_{end_year}
.png')

# Task 5: Maximum melt days and maximum melt extent
max_melt_days = melt_year.max(dim=['y', 'x'],
skipna=True).to_dataframe(name='Max_melt_days')
max_melt_days.index.name = None # Remove index name
max_melt_days['Year'] = max_melt_days.index # extract the index as Year
max_melt_days = max_melt_days.reset_index(drop=True)

# Identify the maximum melt days value and year
max_melt_days_value = max_melt_days['Max_melt_days'].max()
max_melt_days_years = max_melt_days[max_melt_days['Max_melt_days'] ==
max_melt_days_value]['Year'].tolist()
print(f'Max Melt days is {int(max_melt_days_value)} in the following
year/years: {max_melt_days_years}')

# Calculate melt extent (number of pixels with non-NaN values) for each year
in melt_year
melt_year_filtered = melt_year.where(melt_year != 0)
melt_extent = melt_year_filtered.notnull().sum(dim=['y', 'x']) # to create a
boolean mask (.notnull())

# Find the year(s) with the maximum melt extent
max_melt_extent_value = melt_extent.max()
max_melt_extent_years = melt_extent.sel(year=melt_extent ==
```

```
max_melt_extent_value)

print(f'Max Melt Extent is {int(max_melt_extent_value)} in the following
year/years: {", ".join(map(str,
max_melt_extent_years.coords["year"].values))}')

# Task 6: Figures of task 5
# Plot the year with the maximum melt days
for year in max_melt_days_years:
    tmp = melt_year.sel(year=year)
    tmp = tmp.where(tmp != 0, np.nan)

    plot(gdf, lon, lat, tmp,title=f'Max Melt days is
{int(max_melt_days_value)} in {year}',vmax=math.ceil(tmp.max() / 10) *
10,vmin=math.floor(tmp.min() / 10) * 10,save_name=f'Max Melt days is
{int(max_melt_days_value)} in {year}.png')

# Plot the difference of annual average melt days and the year with the
maximum melt days
for year in max_melt_days_years:  # Iterate through the years in
max_melt_days_years
    tmp = melt_year.sel(year=year)
    tmp = tmp.where(tmp != 0, np.nan)

    difference = avg_melt_days - tmp
    plot(gdf, lon, lat, difference,title=f'Difference to Average Annual Melt
Days in Year {year}',vmax=math.ceil(difference.max() / 10) *
10,vmin=math.floor(difference.min() / 10) *
10,cmap='bwr',save_name=f'Difference to Average Annual Melt Days in Year
{year}.png')

year_value = max_melt_extent_years['year'].values.item()
tmp = melt_year.sel(year=year_value)
tmp = tmp.where(tmp != 0, np.nan)
plot(gdf,lon,lat, tmp,title=f'Largest melt extent is
{int(max_melt_extent_value)} in {year_value}',vmax=math.ceil(tmp.max() / 10)
* 10,vmin=math.floor(tmp.min() / 10) * 10,save_name=f'Largest melt extent is
{int(max_melt_extent_value)} in {year_value}.png')
```

```
# Task 7: Convert your xarray to a Pandas DataFrame
melt_data_flattened = melt.values.reshape(melt.shape[0], -1)
pd_melt_extent = np.nansum(melt_data_flattened > 0, axis=1)
days_of_year = time.dt.dayofyear

# Creating Pandas dataframe
df_melt_extent = pd.DataFrame({"datetime": time['time'],"day_of_year":
days_of_year,"melt_extent": pd_melt_extent})
print(df_melt_extent)

# Task 8: Plot melt extent timeseries
# Adding the 'year_group' column to df_melt_extent
df_melt_extent['year_group'] = df_melt_extent['datetime'].apply(lambda x:
x.year if x.month >= 6 else x.year - 1)
df_melt_extent = df_melt_extent.loc[(df_melt_extent['year_group'] >=
start_year) & (df_melt_extent['year_group'] <= end_year-1)] # added
(end_year-1) to remove the incomplete melt data for the last year
(considering the new year period)

# Add a 'custom_date' column to adjust dates for plotting
def adjust_date(date):
    if date.month >= 6:
        return pd.Timestamp(year=1999, month=date.month, day=date.day)
    else:
        return pd.Timestamp(year=2000, month=date.month, day=date.day) # here
the years should include FEB 29th, otherwise it cannot handle leap year dates

df_melt_extent = df_melt_extent.copy()  # Avoid SettingWithCopyWarning
df_melt_extent['custom_date'] = df_melt_extent['datetime'].apply(adjust_date)

# Calculate the average Extent values by 'custom_date'
average_extent = df_melt_extent.groupby('custom_date')['melt_extent'].mean()

# Plot grey lines for each year
plt.figure(figsize=(12, 6), dpi=300)

for year, group in df_melt_extent.groupby('year_group'):
    color = 'red' if year == year_value else 'grey'
```

```
    alpha = 1.0 if year == year_value else 0.5
    label = f"Max Melt Extent ({year})" if color == 'red' else None  # Label
the red line
    plt.plot(group['custom_date'], group['melt_extent'], color=color,
alpha=alpha, label=label)

# Plot the average extent as a black line
plt.plot(average_extent.index, average_extent.values, color='black',
linewidth=2, label=f'Average Melt Extent for {start_year}-{end_year} period')

# Customize the plot
plt.title(f'Melt Extent Data for {start_year}-{end_year} period')
plt.ylabel('Melt Extent / (Pixels)')
plt.grid(True, linestyle='--', alpha=0.6)

# Format the x-axis to show months
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%b'))
plt.gca().xaxis.set_major_locator(mdates.MonthLocator())
plt.legend()

# Save and show the plot
plt.savefig(f'Melt Extent for {start_year}-{end_year} period', dpi=300)
plt.show()

# Task 9: 15 largest ice shelves and average melt extent
gdf = gdf.to_crs(epsg=3031)
gdf["area_calculated"] = gdf.geometry.area
sorted_gdf = gdf.sort_values("area_calculated", ascending=False)
top_15_gdf = sorted_gdf.head(15) # change the number as your interested no of
top ice shelves

melt = melt.rio.write_crs("EPSG:3031", inplace=True)

# Create an empty list to store results
results = []

# Loop through all areas in 'top_15_gdf'
for area_name in top_15_gdf['NAME']:
```

```
    # Select the region for the current area
    region = top_15_gdf[top_15_gdf["NAME"] == area_name]

    # Clip the data for the current region
    clipped = melt.rio.clip(region.geometry, region.crs)

    # Mask the clipped data where the values are equal to 1
    melt_masked = clipped.where(clipped == 1)

    # Resample the data by year and calculate the mean
    yearly_average_melt = melt_masked.resample(time="1YE").mean()
    yearly_average_melt['time'] = yearly_average_melt['time'].dt.year
    yearly_average_melt = yearly_average_melt.rename({"time": "year"})

    # Sum the values over the spatial dimensions x and y
    yearly_sum = yearly_average_melt.sum(dim=["x", "y"])

    # Convert to pandas DataFrame and calculate the mean of the yearly sums
    df_yearly_sum = yearly_sum.to_pandas()
    average_yearly_sum = df_yearly_sum.mean()

    # Append the results (area name and average yearly sum) to the list
    results.append({'NAME': area_name, 'average_yearly_melt_extent':
average_yearly_sum})

# Convert the list of results into a pandas DataFrame
df_top_15_melt = pd.DataFrame(results)

# Print and save the DataFrame
print(df_top_15_melt)
df_top_15_melt.to_csv(f'df_top_15_melt_{start_year}to{end_year}.csv',
index=False)

# Plotting a pie chart for the results
plt.figure(figsize=(8, 8),dpi=300)

# Create the pie chart
plt.pie(df_top_15_melt['average_yearly_melt_extent'],
```

```
labels=df_top_15_melt['NAME'], autopct='%1.1f%%', startangle=90)
plt.title(f'Average Melting Extent for the Largest 15 Ice Shelves from
{start_year} to {end_year}', pad=20)
plt.axis('equal')  # Equal aspect ratio ensures that pie chart is drawn as a
circle.
plt.savefig(f'Average Melting Extent for the Largest 15 Ice Shelves from
{start_year} to {end_year}.png', bbox_inches='tight')  # Save as PNG file
plt.show()
print('/// End of the program ///')
```

6.2 GUI Development Script

```
# Introducing the packages

# Array package
import numpy as np
import pandas as pd
import xarray as xr
# For mathematical functions
import math
# Shapefile loading package
import geopandas as gpd
# Projection plot package
import cartopy as crs
import cartopy.crs as ccrs
from cartopy.feature import ShapelyFeature
# Plotting package
import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
from matplotlib.patches import Patch
from matplotlib_scalebar.scalebar import ScaleBar
import matplotlib.dates as mdates
# Tkinter package
import tkinter as tk
from tkinter import filedialog, messagebox
from PIL import Image, ImageTk
```

```
# Defining the plotting function
def plot(shapefile, lon, lat, data, **kwargs):
    plt.rcParams['font.family'] = 'Times New Roman'

    title = kwargs.get('title', None)
    vmax = kwargs.get('vmax', None)
    vmin = kwargs.get('vmin', None)
    cmap = kwargs.get('cmap', 'Reds')
    save_name = kwargs.get('save_name', None)

    num_levels = 20
    levels = np.linspace(vmin, vmax, num_levels + 1)
    norm = BoundaryNorm(levels, ncolors=256)

    fig = plt.figure(figsize=(6, 6), dpi=100)
    ax = fig.add_subplot(111, projection=ccrs.SouthPolarStereo())
    ax.set_extent([-180, 180, -90, -65], ccrs.PlateCarree())
    ax.add_feature(crs.feature.OCEAN, facecolor='#7cb0d3')
    ax.gridlines(ls='--', color='grey', lw=0.5)

    shape_feature = ShapelyFeature(shapefile.geometry,
ccrs.SouthPolarStereo(), edgecolor='k', facecolor='white', lw=.5)
    ax.add_feature(shape_feature)

    cs = ax.pcolormesh(lon, lat, data, norm=norm, cmap=cmap,
transform=crs.crs.PlateCarree())
    cb = fig.colorbar(cs, ticks=np.linspace(vmin, vmax, 11),
orientation='horizontal', shrink=0.7, pad=0.05)
    cb.ax.set_xlabel('Melt days', fontsize=10)
    ax.set_title(title, weight='bold', fontsize=18, pad=10)

    plt.plot([-59.9, -60], [-59.9, -60], c='k', label='Coastline',
transform=crs.crs.PlateCarree())
    plt.legend(fontsize=10)
    continent_patch = Patch(facecolor='white', edgecolor='black',
label='Antarctic Ice Shelfs')
    sea_patch = Patch(facecolor='#7cb0d3', edgecolor='black', label='Ocean')
    plt.legend(handles=[continent_patch, sea_patch], fontsize=10, loc='lower
```

```
left', frameon=True, framealpha=1, edgecolor='black', fancybox=False)

    scalebar = ScaleBar(1, location='lower right', length_fraction=0.1)
    ax.add_artist(scalebar)

    if save_name:
        fig.savefig(save_name, dpi=100, bbox_inches='tight')

    plt.tight_layout()
    plt.show()
    plt.close()

# Tkinter GUI setup
root = tk.Tk()
root.title('BESUAL Melt Data Analyser')

# Global variables
data = None
gdf = None
start_year = None
end_year = None

def show_instructions():
    # Create a new window for instructions
    instructions_window = tk.Toplevel(root)
    instructions_window.title('Help')

    # Add a label with your instructions text
    instructions_text = '''
    Welcome to the BESUAL Melt Data Analyzer!

    This GUI allows you to analyze and visualize cumulative melt data from
Antarctic regions. Follow the steps below to perform tasks:

    1. Load NetCDF File:
       - Click 'Load NetCDF' to upload your melt data in NetCDF format.

    2. Load Shapefile:
```

```
       - Click 'Load Shapefile' to load a compatible shapefile of your study
region.

    3. Set Analysis Years:
       - Enter the start and end years to filter the dataset for the desired
time range.

    4. Calculate Annual Average Cumulative Melt Days:
       - Compute the annual average of cumulative melt days over the selected
period.

    5. Calculate the Year of Max Melt Days:
       - Identify and visualize the year with the highest cumulative melt
days.

    6. Calculate the Year of Max Melt Extent:
       - Identify the year with the highest melt extent.

    7. Difference of Max Melt Days with Annual Average:
       - Plot the difference between the annual average melt days and the
melt days of the year with maximum melt days.

    8. Plot Melt Extent Timeseries:
       - Examine trends in melt extent across the selected period, with both
individual year and average data visualized.

    9. Melting Data of Top 15 Ice Shelves:
       - Automatically analyze and rank the 15 largest ice shelves based on
their area and melt data.

    Happy analyzing!
    - Team BESUAL -
    '''
    # Display instructions in a label
    instructions_label = tk.Label(instructions_window,
text=instructions_text, padx=10, pady=10, justify='left')
    instructions_label.pack()
```

```
    # Button to close the instructions window
    close_button = tk.Button(instructions_window, text='Close',
command=instructions_window.destroy)
    close_button.pack(pady=10)

# Task 1: Load the shapefile
def load_shapefile():
    global gdf
    file_path = filedialog.askopenfilename(filetypes=[('Shapefiles',
'*.shp')])
    if file_path:
        shapefile_label.config(text=file_path)
        gdf = gpd.read_file(file_path)
        messagebox.showinfo('File Loaded', 'Shapefile loaded successfully!')

# Task 2: Load the NetCDF file
def load_netcdf():
    global data
    file_path = filedialog.askopenfilename(filetypes=[('NetCDF Files',
'*.nc')])
    if file_path:
        netcdf_label.config(text=file_path)
        data = xr.open_dataset(file_path)
        messagebox.showinfo('File Loaded', 'NetCDF file loaded
successfully!')

# Assign start and end years
def set_years():
    try:
        global start_year, end_year, filtered_data, lon, lat, melt, time

        # Get input years
        start_year = int(start_year_entry.get())
        end_year = int(end_year_entry.get())

        # Get the range of years in the NetCDF time variable
        min_year = pd.to_datetime(data['time'].values.min()).year
        max_year = pd.to_datetime(data['time'].values.max()).year
```

```
        # Validate the input years
        if start_year < min_year or end_year > max_year:
            messagebox.showerror('Date Range Error', f'Year range out of
bounds. Valid range: {min_year} to {max_year}.')
            return

        # Filter data based on the years
        filtered_data = data.sel(time=slice(f'{start_year}-01-01',
f'{end_year}-12-31'))

        # Extract variables
        lon = filtered_data['lon'].values if 'lon' in filtered_data else
filtered_data['x'].values
        lat = filtered_data['lat'].values if 'lat' in filtered_data else
filtered_data['y'].values
        melt = filtered_data['melt']
        time = filtered_data['time']

        # Confirmation message
        messagebox.showinfo('Years Set', f'Start year: {start_year}, End
year: {end_year} \nReady for calculations')
    except ValueError:
        # Handle invalid year input
        messagebox.showerror('Input Error', 'Please enter valid years.')

# Task 4: Process data for annual average melt
def annual_average_melt():
    try:
        if data is None or gdf is None:
            messagebox.showerror('Error', 'Please load both NetCDF and
shapefile files.')
            return

        # Group data by year
        melt_year = melt.groupby('time.year').sum(dim='time')

        # Calculate annual average melt
```

```
        avg_melt_days = melt_year.mean(dim='year', skipna=True)
        avg_melt_days = avg_melt_days.where(avg_melt_days != 0, np.nan)

        # Call your plot function
        plot(gdf, lon, lat, avg_melt_days,title=f'Annual Average of
Cumulative Melt Days ({start_year}-
{end_year})',vmax=math.ceil(avg_melt_days.max() / 10) *
10,vmin=math.floor(avg_melt_days.min() / 10) *
10,save_name=f'Annual_Average_of_Cumulative_Melt_Days_{start_year}_{end_year}
.png')

        messagebox.showinfo('Processing Complete', 'Annual average cumulative
melt days plotted successfully.')

    except Exception as e:
        messagebox.showerror('Error', f'An error occurred: {e}')

# Task 5/6: Process max melt days
def max_melt_days():
    try:
        if data is None or gdf is None:
            messagebox.showerror('Error', 'Please load both NetCDF and
shapefile files.')
            return

        melt_year = melt.groupby('time.year').sum(dim='time')

        max_melt_days = melt_year.max(dim=['y', 'x'],
skipna=True).to_dataframe(name='Max_melt_days')
        max_melt_days_value = max_melt_days['Max_melt_days'].max()
        max_melt_days_years = max_melt_days[max_melt_days['Max_melt_days'] ==
max_melt_days_value].index.tolist()

        for year in max_melt_days_years:
            tmp = melt_year.sel(year=year)
            tmp = tmp.where(tmp != 0, np.nan)

            plot(gdf, lon, lat, tmp,title=f'Max Melt days is
```

```
{int(max_melt_days_value)} in {year}',vmax=math.ceil(tmp.max() / 10) *
10,vmin=math.floor(tmp.min() / 10) * 10,save_name=f'Max Melt days is
{int(max_melt_days_value)} in {year}.png')

            messagebox.showinfo('Max Melt Days', f'Max Melt days is
{int(max_melt_days_value)} in {year}')

    except Exception as e:
        messagebox.showerror('Error', f'An error occurred: {e}')

# Task 5/6: Process max melt extent
def max_melt_extent():
    try:
        if data is None or gdf is None:
            messagebox.showerror('Error', 'Please load both NetCDF and
shapefile files.')
            return

        melt_year = melt.groupby('time.year').sum(dim='time')

        # Calculate melt extent (number of pixels with non-NaN values) for
each year in melt_year
        melt_year_filtered = melt_year.where(melt_year != 0)
        melt_extent = melt_year_filtered.notnull().sum(dim=['y', 'x'])

        # Find the year(s) with the maximum melt extent
        max_melt_extent_value = melt_extent.max()
        max_melt_extent_years = melt_extent.sel(year=melt_extent ==
max_melt_extent_value)

        year_value = max_melt_extent_years['year'].values.item()
        tmp = melt_year.sel(year=year_value)
        tmp = tmp.where(tmp != 0, np.nan)
        plot(gdf,lon,lat, tmp,title=f'Largest melt extent is
{int(max_melt_extent_value)} in {year_value}',vmax=math.ceil(tmp.max() / 10)
* 10,vmin=math.floor(tmp.min() / 10) * 10,save_name=f'Largest melt extent is
{int(max_melt_extent_value)} in {year_value}.png')
```

```
        messagebox.showinfo('Max Melt Extent', f'Max Melt Extent is
{int(max_melt_extent_value)} in the following year/years: {", ".join(map(str,
max_melt_extent_years.coords["year"].values))}')

    except Exception as e:
        messagebox.showerror('Error', f'An error occurred: {e}')

# Plot the difference of annual average melt days and the year with the
maximum melt days
def diff_melt_days():
    try:
        if data is None or gdf is None:
            messagebox.showerror('Error', 'Please load both NetCDF and
shapefile files.')
            return

        # Identify the maximum melt days value and year
        melt_year = melt.groupby('time.year').sum(dim='time')
        max_melt_days = melt_year.max(dim=['y', 'x'],
skipna=True).to_dataframe(name='Max_melt_days')
        max_melt_days_value = max_melt_days['Max_melt_days'].max()
        max_melt_days_years = max_melt_days[max_melt_days['Max_melt_days'] ==
max_melt_days_value].index.tolist()

        # Group data by year within the preferred time period
        melt_year = melt.groupby('time.year').sum(dim='time')
        avg_melt_days = melt_year.mean(dim='year', skipna=True)

        for year in max_melt_days_years:  # Iterate through the years in
max_melt_days_years
            tmp = melt_year.sel(year=year)
            tmp = tmp.where(tmp != 0, np.nan)

            difference = avg_melt_days - tmp
            plot(gdf, lon, lat, difference,title=f'Difference to Average
Annual Melt Days in Year {year}',vmax=math.ceil(difference.max() / 10) *
10,vmin=math.floor(difference.min() / 10) *
10,cmap='bwr',save_name=f'Difference to Average Annual Melt Days in Year
```

```
{year}.png')

            messagebox.showinfo('Difference', f'Difference to Average Annual
Melt Days in Year {year} plotted successfully')

    except Exception as e:
        messagebox.showerror('Error', f'An error occurred: {e}')

# Task 8: Plot melt extent
def plot_melt_extent():
    try:
        if data is None or gdf is None:
            messagebox.showerror('Error', 'Please load both NetCDF and
shapefile files.')
            return

        melt_data_flattened = melt.values.reshape(melt.shape[0], -1)
        melt_extent = np.nansum(melt_data_flattened > 0, axis=1)
        days_of_year = time.dt.dayofyear

        # Creating Pandas dataframe
        df_melt_extent = pd.DataFrame({'datetime':
time['time'],'day_of_year': days_of_year,'melt_extent': melt_extent})

        # Adding the 'year_group' column to df_melt_extent
        df_melt_extent['year_group'] =
df_melt_extent['datetime'].apply(lambda x: x.year if x.month >= 6 else x.year
- 1)
        df_melt_extent = df_melt_extent.loc[(df_melt_extent['year_group'] >=
start_year) & (df_melt_extent['year_group'] <= end_year)]

        # Add a 'custom_date' column to adjust dates for plotting
        def adjust_date(date):
            if date.month >= 6:
                return pd.Timestamp(year=1979, month=date.month,
day=date.day)
            else:
                return pd.Timestamp(year=1980, month=date.month,
```

```
day=date.day)

        df_melt_extent = df_melt_extent.copy()  # Avoid
SettingWithCopyWarning
        df_melt_extent['custom_date'] =
df_melt_extent['datetime'].apply(adjust_date)

        # Calculate the average Extent values by 'custom_date'
        average_extent =
df_melt_extent.groupby('custom_date')['melt_extent'].mean()

        # Plot grey lines for each year
        plt.figure(figsize=(12, 6))

        # Variable to define the year with the largest extent
        melt_year = melt.groupby('time.year').sum(dim='time')
        melt_year_filtered = melt_year.where(melt_year != 0)
        melt_extent = melt_year_filtered.notnull().sum(dim=['y', 'x'])
        max_melt_extent_value = melt_extent.max()
        max_melt_extent_years = melt_extent.sel(year=melt_extent ==
max_melt_extent_value)
        max_melt_extent_year_value =
int(max_melt_extent_years['year'].values[0])

        for year, group in df_melt_extent.groupby('year_group'):
            color = 'red' if year == max_melt_extent_year_value else 'grey'
            alpha = 1.0 if year == max_melt_extent_year_value else 0.5
            label = f'Max Melt Extent ({year})' if color == 'red' else None
# Label the red line
            plt.plot(group['custom_date'], group['melt_extent'], color=color,
alpha=alpha, label=label)

        # Plot the average extent as a black line
        plt.plot(average_extent.index, average_extent.values, color='black',
linewidth=2, label=f'Average Melt Extent for {start_year}-{end_year} period')

        # Customize the plot
        plt.title(f'Melt Extent Data for {start_year}-{end_year} period')
```

```
        plt.ylabel('Melt Extent / (Pixels)')
        plt.grid(True, linestyle='--', alpha=0.6)

        # Format the x-axis to show months
        plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%b'))
        plt.gca().xaxis.set_major_locator(mdates.MonthLocator())
        plt.legend()

        # Save and show the plot
        plt.savefig(f'Melt Extent for {start_year}-{end_year} period',
dpi=100)
        plt.show()

        messagebox.showinfo('Melt Extent', f'Melt Extent for {start_year}-
{end_year} period plotted successfully')

    except Exception as e:
        messagebox.showerror('Error', f'An error occurred: {e}')

# Task 9: 15 largest ice shelves and average melt extent
def top_15():
    try:
        if data is None or gdf is None:
            messagebox.showerror('Error', 'Please load both NetCDF and
shapefile files.')
            return

        gdf2 = gdf.to_crs(epsg=3031)
        gdf2['area_calculated'] = gdf2.geometry.area
        sorted_gdf = gdf2.sort_values('area_calculated', ascending=False)
        top_15_gdf = sorted_gdf.head(15)

        melt2 = melt.rio.write_crs('EPSG:3031', inplace=True)

        # Create an empty list to store results
        results = []

        # Loop through all areas in 'top_15_gdf'
```

```
        for area_name in top_15_gdf['NAME']:
            # Select the region for the current area
            region = top_15_gdf[top_15_gdf['NAME'] == area_name]

            # Clip the data for the current region
            clipped = melt2.rio.clip(region.geometry, region.crs)

            # Mask the clipped data where the values are equal to 1
            melt_masked = clipped.where(clipped == 1)

            # Resample the data by year and calculate the mean
            yearly_average_melt = melt_masked.resample(time='1YE').mean()

            # Sum the values over the spatial dimensions x and y
            yearly_sum = yearly_average_melt.sum(dim=['x', 'y'])

            # Convert to pandas DataFrame and calculate the mean of the
yearly sums
            df_yearly_sum = yearly_sum.to_pandas()
            average_yearly_sum = df_yearly_sum.mean()

            # Append the results (area name and average yearly sum) to the
list
            results.append({'NAME': area_name, 'average_yearly_melt_extent':
average_yearly_sum})

        # Convert the list of results into a pandas DataFrame
        df_top_15_melt = pd.DataFrame(results)

        # Print and save the DataFrame
        print(df_top_15_melt)
        df_top_15_melt.to_csv(f'df_top_15_melt_{start_year}to{end_year}.csv',
index=False)

        # Plotting a pie chart for the results
        plt.figure(figsize=(8, 8), dpi=80)

        # Create the pie chart
```

```
        plt.pie(df_top_15_melt['average_yearly_melt_extent'],
labels=df_top_15_melt['NAME'], autopct='%1.1f%%', startangle=90)
        plt.title(f'Average Melting Extent for the Largest 15 Ice Shelves
from {start_year} to {end_year}', pad=20)
        plt.axis('equal')  # Equal aspect ratio ensures that pie chart is
drawn as a circle.
        plt.savefig(f'Average Melting Extent for the Largest 15 Ice Shelves
from {start_year} to {end_year}.png', bbox_inches='tight')  # Save as PNG
file
        plt.show()

        messagebox.showinfo('Melting Extent for the Largest 15 Ice Shelves',
f'Average Melting Extent for the Largest 15 Ice Shelves from {start_year} to
{end_year} plotted successfully and data exported to
df_top_15_melt_{start_year}to{end_year}.csv.')

    except Exception as e:
        messagebox.showerror('Error', f'An error occurred: {e}')

def on_closing():
    # Show a thank-you message when the window is closed
    messagebox.showinfo('Thank You', 'Thank you for using the application!')
    root.destroy()  # Destroy the root window and close the application

# Tkinter Frame
frame = tk.Frame(root)
frame.config(width=750, height=420)  # Adjust as needed
frame.pack(padx=10, pady=10)

# Buttons
help_button = tk.Button(root, text='Help', command=show_instructions)
help_button.place(x=30, y=405)

load_netcdf_button = tk.Button(frame, text='Load NetCDF File',
command=load_netcdf)
load_netcdf_button.place(x=20, y=10)

load_shapefile_button = tk.Button(frame, text='Load Shapefile',
```

```
command=load_shapefile)
load_shapefile_button.place(x=20, y=50)

set_year_button = tk.Button(frame, text='Set Years', command=set_years)
set_year_button.place(x=480, y=90)

task4_button = tk.Button(frame, text='Calculate Annual Average Cumulative
Melt Days', command=annual_average_melt)
task4_button.place(x=350, y=150)

task5a_button = tk.Button(frame, text='Calculate the Year of Max Melt Days',
command=max_melt_days)
task5a_button.place(x=350, y=190)

task5b_button = tk.Button(frame, text='Calculate the Year of Max Melt
Extent', command=max_melt_extent)
task5b_button.place(x=350, y=230)

task6b_button = tk.Button(frame, text='Difference of Max Melt Days with
Annual Average', command=diff_melt_days)
task6b_button.place(x=350, y=270)

task8_button = tk.Button(frame, text='Plot Melt Extent Timeseries',
command=plot_melt_extent)
task8_button.place(x=350, y=310)

task9_button = tk.Button(frame, text='Melting Data of Top 15 Ice Shelves',
command=top_15)
task9_button.place(x=350, y=350)

# Add a label to display the file path
netcdf_label = tk.Label(frame, text='No file selected', fg='black',
anchor='w')
netcdf_label.place(x=150, y=15)  # Adjust the `x, y` coordinates as needed

shapefile_label = tk.Label(frame, text='No file selected', fg='black',
anchor='w')
shapefile_label.place(x=150, y=55)  # Adjust the `x, y` coordinates as needed
```

```
start_year_label = tk.Label(root, text='Start Year:')  # Directly in the root
window
start_year_label.place(x=40, y=105)  # Adjust coordinates relative to the
window
start_year_entry = tk.Entry(root)  # Directly in the root window
start_year_entry.place(x=100, y=105)  # Adjust coordinates relative to the
window

end_year_label = tk.Label(root, text='End Year:')
end_year_label.place(x=250, y=105)
end_year_entry = tk.Entry(root)
end_year_entry.place(x=310, y=105)

# Load the image using Pillow
image_path = 'sit_antarctica_map.png'  # Replace with the path to your image
image = Image.open(image_path)
# Resize the image if needed
image = image.resize((300, 250))  # Resize the image to fit the GUI
# Convert the image to Tkinter format
image_tk = ImageTk.PhotoImage(image)
# Create a label to display the image
image_label = tk.Label(root, image=image_tk)
image_label.place(x=30, y=150)

logo_path = 'logo.webp'  # Replace with the path to your image
logo = Image.open(logo_path)
# Resize the image if needed
logo = logo.resize((110, 110))  # Resize the image to fit the GUI
# Convert the image to Tkinter format
logo_tk = ImageTk.PhotoImage(logo)
# Create a label to display the image
logo_label = tk.Label(root, image=logo_tk)
logo_label.place(x=650, y=15)

root.protocol('WM_DELETE_WINDOW', on_closing)

root.mainloop()
```

6.3 Support from Online Resources

The development of the code was significantly supported by online resources, particularly the Generative Pre-training Transformer (GPT). This tool was utilized extensively to address syntax errors, which constituted its primary use during this process. Additionally, GPT was instrumental in simplifying complex logical structures, enabling us to determine optimal solutions while reducing computational power and execution time requirements. These optimized solutions were seamlessly integrated into the code during development.

Moreover, AI assistance played a vital role in creating the foundational setup of the Graphical User Interface (GUI). We used AI to design the initial framework and then incorporated the custom functions developed in the basic script into the GUI. Following this integration, we applied necessary adjustments, and finalized the script.

Beyond AI tools, we relied heavily on community-driven platforms such as Stack Overflow to resolve bugs and technical challenges. Python's extensive and active user community provided invaluable insights and solutions through such forums. The contributions from these resources were crucial, given the vast amount of shared knowledge and expertise available within these communities.